Science of Computer Programming 2 (1982) 207–214
North-Holland

A NOTE ON A STANDARD STRATEGY FOR DEVELOPING LOOP INVARIANTS AND LOOPS

David GRIES

Department of Computer Science, Cornell University, Ithaca, NY 14853, U.S.A.

Communicated by M. Sintzoff
Received December 1982
Revised March 1983

Abstract. A by-now-standard strategy for developing a loop invariant and loop was developed in [1] and explained in [2]. Nevertheless, its use still poses problems for some. The purpose of this paper is to provide further explanation. Two problems are solved that, without this further explanation, seem difficult.

1. Introduction

A standard strategy for developing a loop invariant and loop can indeed be powerful in the hands of one who understands it fully. Those with less experience, however, have difficulty applying it in the seemingly complex situations where it is really needed. This may be due, at least partially, to the way the strategy is presented. The purpose of this note is to present the strategy in a slightly different and less formal manner.

We begin by discussing the proof of correctness of a loop. We then develop a rather trivial algorithm in order to describe the strategy in its simplest form. Finally, we proceed to illustrate a more difficult part of the strategy using two examples.

2. Proving a loop correct

E.W. Dijkstra's guarded command loop with precondition Q and postcondition R, $\{Q\}$ **do** $B \rightarrow S$ **od** $\{R\}$, is proved correct using an *invariant relation* P and a *bound function* t, which gives an upper bound on the number of iterations still to perform (see e.g. [1, 2]):

(1) P is true initially: $Q \Rightarrow P$;

(2) P is a loop invariant: $P \wedge B \Rightarrow wp(S, P)$;

(3) Upon termination R is true: $P \wedge \neg B \Rightarrow R$;

(4) If another iteration can be performed, then $t > 0$: $P \wedge B \Rightarrow t > 0$;

(5) t is decreased by at least one with each iteration: using a fresh variable *t1*, we have

$$\{P \wedge B\}\ t1 := t;\ S\ \{t < t1\}.$$

3. Developing a loop using the standard strategy

To illustrate the strategy for developing a loop, let us develop an algorithm for storing in variable p the sum of the elements of array $b[0: n-1]$, where $0 \leq n$. The postcondition of the algorithm is

$$R: \quad p = (\textstyle\sum k: 0 \leq k < n: b[k]).$$

Recognizing the need for iteration (or recursion), we try to develop a loop invariant P first—by generalizing R to include a state that can be easily established. In this case, generalizing R can be done by replacing constant n of R by a fresh variable i and placing suitable bounds on i:

$$P: \quad 0 \leq i \leq n \wedge p = (\textstyle\sum k: 0 \leq k < i: b[k]).$$

P is easily established using $i, p := 0, 0$. Further, $(P \wedge i = n) \Rightarrow R$, so we can take $i \neq n$ as the guard of the loop.

The bound function t, an upper bound on the number of iterations still to perform, is $n - i$. To reduce the bound function at each iteration, choose the command $i := i + 1$, yielding, thus far,

```
$i, p := 0, 0;$
do $i \neq n \rightarrow \cdots i := i + 1$ od
```

The last step is to ensure that P is indeed a loop invariant, i.e. $(P \wedge i \neq n) \Rightarrow wp(i := i + 1, P)$. To do this, we first calculate $wp(i := i + 1, P)$ and rearrange it to prepare for comparison with $P \wedge i \neq n$:

$$0 \leq i + 1 \leq n \wedge p = \textstyle\sum (k: 0 \leq k < i + 1: b[k])$$
$$= 0 \leq i + 1 \leq n \wedge p = \textstyle\sum (k: 0 \leq k < i: b[k]) + b[i]$$

Comparing this predicate with P, we see that adding $b[i]$ to p within the loop body will allow P to remain invariantly true, and we end up with the algorithm

```
$i, p := 0, 0;$
do $i \neq n \rightarrow p := p + b[i]; i := i + 1$ od
```

The steps in this development are fairly standard, and we discuss only those pertinent to this paper, i.e. those dealing with the development of invariant P. The first step was the following:

(6) Find the (first approximation to the) invariant by generalizing the postcondition R.

Some techniques for generalizing a predicate are discussed in [1] and [2].

The second point is that once a way is found to reduce the bound function—in our case $i := i + 1$—the invariance of P must be shown: for S the loop body determined so far, $P \wedge B \Rightarrow wp(S, P)$ must be true. If not, S must be modified, and to determine the modification one investigates the difference between $P \wedge B$ and $wp(S, P)$.

This part of the strategy can be described as follows:

(7) Determine the conditions under which decreasing the bound function will falsify the invariant, and modify the loop body S to prevent such falsification.

Note that a loop developed strictly in this fashion has the form

```
        {invariant: $P$, bound function: $t$}
        do $B \rightarrow \{P \wedge B\}$
                Establish $wp$('Decrease $t$', $P$);
                {$wp$('Decrease $t$', $P$)}
                Decrease $t$
                {$P$}
        od
```

Steps (6) and (7) are often easy to apply. However, in some cases applying (7) may seem difficult or may lead to inefficient and clumsy algorithms. At this point, more direction is needed: this direction is summarized as follows:

(8) Determine what further information is needed in order to make application of (7) more effective, and represent that information in fresh variables, thus modifying P.

It is essential that the further information to be stored in fresh variables can be extracted from the 'scanned portion' of the variables, so that it can be extracted as one goes along.

Conscious application of (8) can indeed be useful. To illustrate this, let us turn to two 'difficult' problems. For purposes of comparison, the reader may want to try developing the algorithms using her own methods before reading the idealized developments.

4. A first example: The minimum-sum section

A *minimum-sum section* of an array b is a non-empty sequence of adjacent elements whose sum is a minimum. For example, the minimum-sum section of array $b[0{:}\ 4] = (5, -3, 2, -4, 1)$ is $b[1{:}\ 3] = (-3, 2, -4)$; its sum is -5. The minimum-sum section of array $b = (5, -3, 5, -4, 1)$ is $b[3{:}\ 3] = (-4)$; its sum is -4. The two minimum-sum sections of $b = (5, 2, 5, 4, 2)$ are $b[1{:}\ 1]$ and $b[4{:}\ 4]$.

Desired is a program that, given a nonempty array $b[0{:}\ n-1]$, stores in variable s the sum of a minimum-sum section of b. Let $S_{i,j}$ denote the sum of section $b[i{:}\ j]$:

$$S_{i,j} = (\sum k : i \leq k \leq j : b[k]).$$

Then the postcondition R is

$$R: \quad s = \mathbf{min}(i, j\colon 0 \le i \le j < n\colon S_{i,j}).$$

It is obvious that each element of b must be referenced to determine a minimum-sum section and its sum, and our first thought is to reference them in increasing order of subscript value. Using strategy (6) as our guide, we find a first approximation to the invariant P by replacing in R constant n by a fresh variable k (and placing bounds on k):

$$P: \quad 1 \le k \le n \wedge s = \mathbf{min}(i, j\colon 0 \le i \le j < k\colon S_{i,j}).$$

The initialization is obvious, as is, the bound function $n - k$. And we write

```
$k, s := 1, b[0]$;
do $k \neq n \rightarrow \cdots k := k + 1$ od
```

Next, applying (7), we determine the conditions under which P may be falsified by execution of the loop body. First calculate and rearrange $wp(k := k + 1, P)$. We rearrange the predicate so that it is easy to compare with $P \wedge k \neq n$[1]

$$\begin{aligned} & wp(k := k + 1, P) \\ & \quad = 1 \le k + 1 \le n \wedge s = \mathbf{min}(i, j\colon 0 \le i \le j < k + 1\colon S_{i,j}) \\ & \quad = 1 \le k + 1 \le n \wedge \\ & \qquad s = \mathbf{min}(i, j\colon 0 \le i \le j < k\colon S_{i,j})\, \mathit{min}\, \mathbf{min}(i\colon 0 \le i \le k\colon S_{i,k}) \end{aligned} \tag{9}$$

The first conjunct of (9) is implied by $P \wedge k \neq n$, but the second is not. Comparing the second conjuncts of P and (9), we see that $\neg(P \Rightarrow Q)$ holds precisely when the value

$$\mathbf{min}(i\colon 0 \le i \le k\colon S_{i,k})$$

is less than s—i.e. when a section $b[i\colon k]$ for some i has sum is less than s.

At this point, it looks like determining whether a section $b[i\colon k]$ with $S_{i,k} < s$ exists may take time at least proportional to k. What can we do to make it more efficient? Turning to strategy (8), we ask what additional information can help. Suppose we know $\mathbf{min}(i\colon 0 \le i < k\colon S_{i,k-1})$. Then $\mathbf{min}(i\colon 0 \le i \le S_{i,k})$ can be calculated in constant time: it is the minimum of $\mathbf{min}(i\colon 0 \le i < k\colon S_{i,k-1}) + b[k]$ and $b[k]$. We introduce a variable c and change P accordingly:

$$\begin{aligned} P: \quad & 1 \le k \le n \\ & \wedge s = \mathbf{min}(i, j\colon 0 \le i \le j < k\colon S_{i,j}) \\ & \wedge c = \mathbf{min}(i\colon 0 \le i < k\colon S_{i,k-1}). \end{aligned}$$

[1] Throughout, the minimum and maximum functions *min* and *max* are written as binary infix operators.

Note that P implies that $s \leqslant c$. It is then a simple task to modify and complete the algorithm:

```
$k, s, c := 1, b[0], b[0]$;
do $k \neq n \rightarrow c := min\ c + b[k]b[k]$;
              $s := s\ min\ c$;
              $k := k + 1$
od
```

This algorithm has the following history. Shamos of Carnegie-Mellon University saw a statistician using an $O(n^3)$ algorithm that determined the bounds of a minimum-sum section as well as its sum. Shamos and Jon Bentley developed an $O(n^2)$ algorithm and, about a week later, an $O(n \log n)$ algorithm. Two weeks after having first seen the problem, they discussed it with another statistician, who immediately gave them linear algorithm. Bentley, while discussing programming methodology at Cornell, challenged us with this problem. The linear algorithm was developed basically as shown above—the usual starts and restarts to get familiar with the problem are not shown, and a less formal notation was used—and, in fact, it was the only algorithm considered.

The determination of a minimum-sum section itself has been omitted simply to omit detail not germane to the topic at hand. The reader may find it interesting to solve the same problem with a slight change: empty sections should also be considered. Thus, if the array contains only positive values, the minimum-sum section is the empty section and its sum is 0.

5. A second example: Finding the largest square

We proceed with this problem in the same manner as with the last. We will, however, use a two-dimensional 'picture' notation for part of the invariant, which may make it easier to understand the development. This notation can be translated easily into the predicate calculus.

Given a rectangular, Boolean array $b[0\colon m-1, 0\colon n-1]$, where $m, n > 0$, calculate the size—i.e. length of a side—of a largest square of adjacent true elements. In what follows, we abbreviate 'square of adjacent true elements' by '*tsquare*'. Thus, we write the postcondition R as

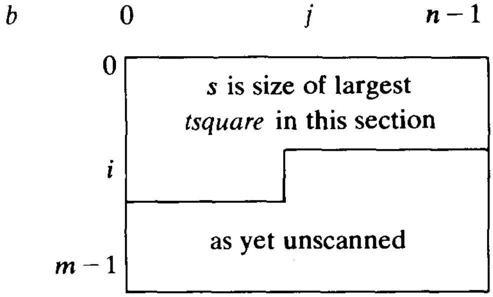
$$R\colon \quad s \text{ is the size of the largest } tsquare \text{ in } b[0\colon m-1, n-1].$$

In the worst case, it will be necessary to reference each element of the array, so we consider ways of referencing each element. Row-by-row traversal is simple, so we generalize R to the invariant $P = P1 \wedge P2$, where

$$P1{:}\quad 0 \leqslant i \leqslant m \wedge 0 \leqslant j < n$$

and

$P2$:



Thus, $b[i, j]$ is the next element to be scanned. The bound function t is the number of elements in the lower, unscanned, section. The obvious way to proceed towards termination is to scan element $b[i, j]$, changing j (and i, if necessary) accordingly. According to strategy (7), we now have the task of maintaining P when $b[i, j]$ is scanned. We first determine the conditions under which P is falsified:

> P is falsified if there is a *tsquare* with lower right corner $b[i, j]$ whose side is longer than s.

Calculating the size of the largest *tsquare* with lower right corner $b[i, j]$ seems to be a fairly complicated task, so we turn to strategy (8) for some inspiration. What useful information can we save in fresh variables? There are several alternatives. Among them are

(1) Maintain the sizes of the largest *tsquares* with lower right corners $b[i, j-1]$ and $b[i-1, j]$—from them it is easy to calculate the size of the largest *tsquare* with lower right corner $b[i, j]$;

(2) Maintain the size of the largest *tsquare* with lower right hand corner $b[i-1, j-1]$, the length of the column of true values ending in $b[i-1, j]$, and the length of the row of true values ending in $b[i, j-1]$.

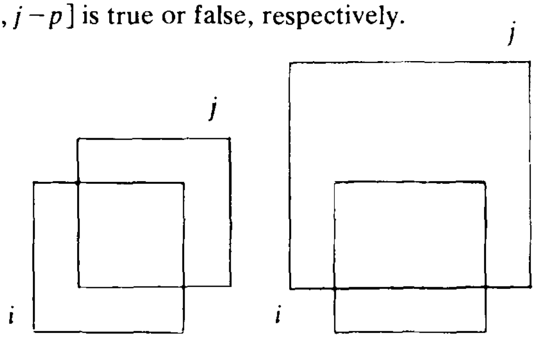
In any case, since the problem will occur with each new column, the information will be needed for each column. Let us implement the first alternative. Add a conjunct $P3$ to the invariant that describes a one-dimensional array $c[-1{:}\, n-1]$, where element $c[-1]$ is introduced to remove a case analysis:

$$P{:}\quad P1 \wedge P2 \wedge P3,$$

where

$$
\begin{aligned}
P3\colon\ & c[-1]=0 \\
& \wedge (\forall k\colon 0\leq k<j\colon c[k] \\
& \qquad = \text{size of largest } \textit{tsquare} \text{ with lower right corner } b[i,k]) \\
& \wedge (\forall k\colon j\leq k<n\colon c[k] \\
& \qquad = \text{size of largest } \textit{tsquare} \text{ with lower right corner } b[i-1,k]).
\end{aligned}
$$

$P3$ must be maintained as $b[i,j]$ is scanned, and thus we must determine how to change $c[j]$. Suppose $b[i,j]$ is true. Consider the three cases $c[j-1]>c[j]$, $c[j-1]=c[j]$ and $c[j-1]<c[j]$, the last two of which are drawn below. (The first case is similar to the last.) Letting $p = c[j-1]\ \mathit{min}\ c[j]$, we see in each case that the size of the largest *tsquare* with lower right corner $b[i,j]$ is either $p+1$ or p, depending on whether $b[i-p, j-p]$ is true or false, respectively.



$c[j-1]=c[j]$ $\qquad$ $c[j-1]<c[j]$

With this information, the algorithm is written as follows:

```
$i, j, s := 0, 0, 0;$
$(\forall k\colon -1\leq k<n\colon c[k]:=0);$
**do** $i\neq m \rightarrow$
    **if** $\neg b[i,j] \rightarrow c[j]:=0$
    □   $b[i,j] \rightarrow$ **var** $p$: integer;
                  $p := c[j-1]\ \mathit{min}\ c[j];$
                  **if** $\neg b[i-p, j-p] \rightarrow c[j]:=p$
                  □   $b[i-p, j-p] \rightarrow c[j]:=p+1$
                  **fi**
    **fi**;
    $\{c[j]$ is size of largest square with lower right corner $b[i,j]\}$
    $s := s\ \mathit{max}\ c[j];$
    **if** $j=n-1 \rightarrow i, j := i+1, 0$
    □ $j<n-1 \rightarrow j:=j+1$
    **fi**
**od**
```

The execution time of this algorithm is $O(m*n)$. Again, this was essentially the only algorithm thought of during the development, although several ways of maintaining the necessary information were thought of. I learned of the problem from Ed Cohen of Prime, but discovered that it, and its appearance in [4], can be traced to W. H. J. Feijen.

6. Discussion

This strategy for developing loop invariants and loops has been used often in the past; in fact, at times it seems the only reasonable way to proceed. A good example of its use is in the development of an algorithm for the longest upsequence problem, due originally to Dijkstra [0], which can also be found in [2]. There, the strategy had to be applied a number of times until it became obvious how the invariant had to be generalized. Reference [3] discusses the same strategy but in a more limited context.

However, many experienced and unexperienced programmers have been unable to solve effectively either of the two problems shown above, partially because they were unable to apply these problem-solving techniques in a conscious manner. Hence, this paper.

Acknowledgment

I am grateful to Fred Schneider for criticizing a draft of this paper and to J. Misra for pointing out reference [3]—a paper I had refereed but forgotten! Thanks also to members of IFIP WG2.3 for comments on a presentation of this material at our Mohonk meeting, to members of class CS600 at Cornell, many of whom were able to solve the two problems in a manner similar to the one shown after having been taught the strategy, and to the Tuesday Afternoon Club in Eindhoven, especially Edsger W. Dijkstra and Netty van Gasteren, for their comments on the problems and the standard techniques and for criticisms of a draft of this paper.

References

[0] E. W. Dijkstra, Some beautiful arguments using mathematical induction, *Acta Informat.* **13** (1980) 1–8.

[1] E.W. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).

[2] D. Gries, *The Science of Programming* (Springer, New York, 1981).

[3] J. Misra, A technique of algorithm construction on sequences, *IEEE Trans. Software Engrg.* **4** (1) (1978) 65–69.

[4] J. Reynolds, *The Craft of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1981).